WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Tuesday, August 24, 2004

Hide?	Set Name	Query	Hit Count
	DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
☐	L6	L5 and TCP	23
☐	L5	20010531	429
☐	L4	L2 same (database adj application)	0
☐	L3	L2 same (application adj software)	1
☐	L2	socket adj control	760
☐	L1	socket adj control adj code	2

END OF SEARCH HISTORY

WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Tuesday, August 24, 2004

Hide?	Set Name	Query	Hit Count
	DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
☐	L11	20010531	5
☐	L10	(database adj3 (application or SQL)) same (winsock or (Berkeley adj socket) or (conventional adj socket))	5
☐	L9	L8 and ((data or file) adj2 transfer)	27
☐	L8	L7 and (application-specific)	32
☐	L7	20010531	228
☐	L6	20010531	793
☐	L5	20010513	774
☐	L4	TCP adj socket	442
☐	L3	TCP adj2 socket	1450
☐	L2	L1	670
	DB=USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
☐	L1	TCP adj2 socket	670

END OF SEARCH HISTORY

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Generate Collection

L9: Entry 21 of 27 File: USPT Apr 16, 2002

DOCUMENT-IDENTIFIER: US 6374299 B1
TITLE: Enhanced scalable distributed network controller

Application Filing Date (1):
20000126

Brief Summary Text (21):
In accordance with the varying aspects of the present invention, system architecture operates in a client-server environment with a plurality of individual services resident on one or more servers in the network. Clients operate within the network and access these services by initiating an instance of the Framework API, which allows network communication to the Framework supported services, associated with the application-specific DLL's necessary to permit access to the service and internetwork communication.

Detailed Description Text (10):
In this arrangement, the Framework acts as a controlling layer negotiating the communications and data transfers between the clients, 100 and the service 10. In this context, the Framework takes on multiple roles, depending on the service DLLs selected and loaded. At the client, 100, the Framework API, 20, loads the modules (DLLs) that control the formatting of the messages, manage outgoing service requests and collect incoming service responses from the network. Similarly, a separate instance of the Framework, 15, is loaded at the service, 10 which invokes the DLLs for coordinating the processing of client requests--communicated to the service by the network--and generating the responses for return to the client 100 via the network.

Detailed Description Text (15):
Continuing with FIG. 3A, the second client server pair is between HTTP server 200 and the network servers 220 (plural servers depicted in this Figure). Focusing on this relationship, the network servers 220 act as Routers (see below) to direct messages to available services. Access to these services are through the system servers 250. One or more instances of the Framework is initiated on server 250 invoking the service DLL associated with the initial client request. As described below, once initiated, the Framework manages the connections and data transfers to support the specific service invoked. For example, the market data service is provided by invocation of the market data DLL by the Framework which then manages the communications to the outside vendor 260. In a similar manner the Framework, by invoking the Mainframe access DLL, supports communication to the Mainframe via SP/2. The above Router servers, 220, depicted in FIG. 3A, provides an illustrative example of Router operation. In this arrangement, the Router operation exists on a server no different than that depicted in block 250, and the Router represents another "service". This service is provided by the invocation of the Router.dll by the Framework, and directs access to the service providers ("services") based on availability.

Detailed Description Text (24):
Test 570 determines whether listening socket is applicable; if so, the TCP socket is created at block 580. Likewise, test 590 checks for UDP reading socket which is

created at block 600 if applicable. In accordance with the above assignments, initial I/O buffers are created. This is accomplished at test 610, and, if required, block 620.

Detailed Description Text (26):
Next, the Framework creates the worker threads; these are set in place to implement the service functionality. The number of worker threads (minimum and maximum) are optional parameters that are stored in the registry. This is followed by the creation of the abstract base class object, block 700 from the DLL (template object X). This object data is then initialize, block 720. Operation depends on the communication. If UDP socket, Framework creates a supporting UDP thread; test 730 and block 740. In a similar way, if TCP socket, a TCP thread accept is created; test 750 and block 760. The Framework may test for both heartbeat information and general administration requests, via tests 770 and 790, respectfully. To the extent active messages on these are expected, associated threads are assigned at blocks 780 and 800. Otherwise, the Framework remains resident until terminated, test 810 and block 830.

Detailed Description Text (36):
If a READ operation, then control flows to block 2302, where a test is made to determine whether the READ operation encountered some kind of error. If so, then block 2304 receives control and closes the TCP socket for which the error occurred, and then recovers the data buffer at block 2306 for use later in other I/O operations. If no error occurred, then block 2308 receives control and tests to determine whether the I/O operation resulted in a full client message being received over the socket. If the test is affirmative, then control is passed to block 2310, which creates a request packet and marks it to indicate a Process TCP request as mentioned in the discussion of FIG. 9. Block 2312 then places the request packet on the end of a queue, and then block 2314 signals the "request ready" signal that all worker threads wait upon (as shown in block 2020 of FIG. 9). If the test for a full message is negative, then the thread must initiate another asynchronous I/O READ operation on the socket in order to receive more data. It does this at block 2316.

Detailed Description Text (37):
If a WRITE operation, then control flows to block 2303, where a test is made to determine whether the WRITE operation encountered some kind of error. If so, then block 2305 receives control and closes the TCP socket for which the error occurred, and then recovers the data buffer at block 2307 for use later in other I/O operations. If no error occurred, then block 2309 receives control and tests to determine whether the I/O operation resulted in a full response message being sent over the socket to the client. If the test is affirmative, then control is passed to block 2311, which simply recovers the data buffer for later use. If the test for a full message is negative, then the thread must initiate another asynchronous I/O WRITE operation on the socket in order to ensure that all date is sent to the waiting client. It does this at block 2313.

Previous Doc Next Doc Go to Doc#

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Search Forms
Search Results
Help
User Searches
Preferences
Logout

Generate Collection

L9: Entry 26 of 27 File: USPT Apr 4, 2000

DOCUMENT-IDENTIFIER: US 6047324 A
TITLE: Scalable distributed network controller

Application Filing Date (1):
19980205

Brief Summary Text (21):
In accordance with the varying aspects of the present invention, system architecture operates in a client-server environment with a plurality of individual services resident on one or more servers in the network. Clients operate within the network and access these services by initiating an instance of the Framework API, which allows network communication to the Framework supported services, associated with the application-specific DLL's necessary to permit access to the service and internetwork communication.

Detailed Description Text (10):
In this arrangement, the Framework acts as a controlling layer negotiating the communications and data transfers between the clients, 100 and the service 10. In this context, the Framework takes on multiple roles, depending on the service DLLs selected and loaded. At the client, 100, the Framework API, 20, loads the modules (DLLs) that control the formatting of the messages, manage outgoing service requests and collect incoming service responses from the network. Similarly, a separate instance of the Framework, 15, is loaded at the service, 10 which invokes the DLLs for coordinating the processing of client requests--communicated to the service by the network--and generating the responses for return to the client 100 via the network.

Detailed Description Text (15):
Continuing with FIG. 3A, the second client server pair is between HTTP server 200 and the network servers 220 (plural servers depicted in this FIG.). Focusing on this relationship, the network servers 220 act as Routers (see below) to direct messages to available services. Access to these services are through the system servers 250. One or more instances of the Framework is initiated on server 250 invoking the service DLL associated with the initial client request. As described below, once initiated, the Framework manages the connections and data transfers to support the specific service invoked. For example, the market data service is provided by invocation of the market data DLL by the Framework which then manages the communications to the outside vendor 260. In a similar manner the Framework, by invoking the Mainframe access DLL, supports communication to the Mainframe via SP/2. The above Router servers, 220, depicted in FIG. 3A, provides an illustrative example of Router operation. In this arrangement, the Router operation exists on a server no different than that depicted in block 250, and the Router represents another "service". This service is provided by the invocation of the Router.dll by the Framework, and directs access to the service providers ("services") based on availability.

Detailed Description Text (24):
Test 570 determines whether listening socket is applicable; if so, the TCP socket is created at block 580. Likewise, test 590 checks for UDP reading socket which is

created at block 600 if applicable. In accordance with the above assignments, initial I/O buffers are created. This is accomplished at test 610, and, if required, block 620.

Detailed Description Text (26):
Next, the Framework creates the worker threads; these are set in place to implement the service functionality. The number of worker threads (minimum and maximum) are optional parameters that are stored in the registry. This is followed by the creation of the abstract base class object, block 700 from the DLL (template object X). This object data is then initialize, block 720. Operation depends on the communication. If UDP socket, Framework creates a supporting UDP thread; test 730 and block 740. In a similar way, if TCP socket, a TCP thread accept is created; test 750 and block 760. The Framework may test for both heartbeat information and general administration requests, via tests 770 and 790, respectfully. To the extent active messages on these are expected, associated threads are assigned at blocks 780 and 800. Otherwise, the Framework remains resident until terminated, test 810 and block 830.

Detailed Description Text (36):
If a READ operation, then control flows to block 2302, where a test is made to determine whether the READ operation encountered some kind of error. If so, then block 2304 receives control and closes the TCP socket for which the error occurred, and then recovers the data buffer at block 2306 for use later in other I/O operations. If no error occurred, then block 2308 receives control and tests to determine whether the I/O operation resulted in a full client message being received over the socket. If the test is affirmative, then control is passed to block 2310, which creates a request packet and marks it to indicate a Process TCP request as mentioned in the discussion of FIG. 9. Block 2312 then places the request packet on the end of a queue, and then block 2314 signals the "request ready" signal that all worker threads wait upon (as shown in block 2020 of FIG. 9). If the test for a full message is negative, then the thread must initiate another asynchronous I/O READ operation on the socket in order to receive more data. It does this at block 2316.

Detailed Description Text (37):
If a WRITE operation, then control flows to block 2303, where a test is made to determine whether the WRITE operation encountered some kind of error. If so, then block 2305 receives control and closes the TCP socket for which the error occurred, and then recovers the data buffer at block 2307 for use later in other I/O operations. If no error occurred, then block 2309 receives control and tests to determine whether the I/O operation resulted in a full response message being sent over the socket to the client. If the test is affirmative, then control is passed to block 2311, which simply recovers the data buffer for later use. If the test for a full message is negative, then the thread must initiate another asynchronous I/O WRITE operation on the socket in order to ensure that all date is sent to the waiting client. It does this at block 2313.

Previous Doc Next Doc Go to Doc#

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Generate Collection

L6: Entry 13 of 23 File: USPT Apr 9, 2002

DOCUMENT-IDENTIFIER: US 6370569 B1
**** See image for Certificate of Correction ****
TITLE: Data socket system and method for accessing data sources using URLs

Application Filing Date (1):
19981103

Brief Summary Text (14):
The Data Socket client is preferably a Data Socket control or other type of re-useable software component. The Data Socket client includes a simple set of properties which provides ease of use. The Data Socket client includes a property referred to as URL which is used to specify the data source. The Data Socket client also includes a status property which informs the program using the Data Socket client as to the state of the connection, i.e., whether the socket is unconnected or connected.

Brief Summary Text (17):
Once the URL is provided, the Data Socket client examines the URL and determines if the Data Socket control can process the data. If so, the Data Socket client retrieves the data. During the data retrieval, the Data Socket client handles opening and closing of the file. The Data Socket client also converts the data to a format useable by the application. In the preferred embodiment, the Data Socket client obtains the data and converts it to a format referred to as a Flex Data object. Once the data has been received and converted, the Data Socket client generates an event to the application program, and the application can receive and process the data.

Brief Summary Text (20):
The present invention also includes a new data format referred to as Flex Data or FlexData, as mentioned above. Flex Data is the format of the underlying data packet that Data Socket tools use to send and store information. Flex Data data packets are system independent and can be transferred over existing communication technologies such as ActiveX Automation/COM or TCP/IP. The data packet has provisions for adding standard and user defined named attributes that are used to further describe the information contained in the packet. Information such as a time stamp, test operator name, Unit-Under-Test (UUT) identifier, and more can be added to the data being transferred. Data Socket clients convert data native to an environment and its attributes into Flex Data format or convert Flex Data back into its native data form. The Flex Data packet is self-describing so no additional information must be exchanged between a client and its source or target.

Detailed Description Text (33):
Data Socket tools allow users to easily transfer instrumentation style data between different applications, applications and files, as well as different machines. Currently these types of transfer are done using less specialized tools such as general-purpose file I/O functions, TCP/IP functions, HTTP/HTML transfers combined with CGI programs and more. With all of these existing technologies, the user is required to perform a significant amount of programming to accomplish his/her tasks. The Data Socket client and tools of the present invention provide improved

functionality and reduce development effort and time.

Detailed Description Text (38):
TCP/IP

Detailed Description Text (39):
TCP/IP (transfer control protocol/Internet protocol) is used to transfer data between the DataServer and different Data Socket clients.

Detailed Description Text (48):
The Data Socket client, also referred to as the Data Socket control, is the name used to refer to the component that an application uses to connect to a Data Socket source or target. Data Socket clients have different formats for different development environments. (ActiveX control, LabVIEW VI, CVI instrument driver/DLL, etc.)

Detailed Description Text (50):
Flex Data, also referred to as FlexData, is the format of the underlying data packet that Data Socket tools use to send and store information. These packets are system independent and can be transferred over existing communication technologies such as ActiveX Automation/COM or TCP/IP. The Flex Data data packet has provisions for adding standard and user defined named attributes that are used to further describe the information contained in the packet. For instrumentation-specific functions, information such as a time stamp, test operator name, Unit-Under-Test (UUT) identifier, and more can be added to the data being transferred. Data Socket clients convert data native to an environment and its attributes into Flex Data format or convert Flex Data formatted data back into its native data form. The Flex Data packet is self-describing so no additional information must be exchanged between a client and its source or target.

Detailed Description Text (57):
The Data Socket client functionality is preferably implemented with an ActiveX control and underlying objects. The ActiveX control is preferably optimized for Microsoft Visual Basic/VBA(Visual Basic for Applications)/VB Script but is also compatible with other ActiveX control containers including Microsoft Visual C/C++, Borland Delphi, Borland C++ Builder and National Instruments HiQ 4.0. The Data Socket ActiveX control is part of the ComponentWorks tools suite available from National Instruments Corporation and is called CWData Socket. One other significant ActiveX component is the CWData object which stores data transferred by the CWData Socket control.

Detailed Description Text (66):
The Data Socket client includes built-in support for standard sources such as files, FTP and HTTP. Other built-in support includes exchange with a Data Socket DataServer using TCP/IP, referred to as DSTP. Developers are also able to define their own extensions or plug-ins that allow the Data Socket to connect to their own data sources. Extensions are identified by the URL access method. The URL Access is the part of the URL string that precedes the first colon ":". A Data Socket extension is preferably implemented using an automation server written to a common specification. The automation server and unique extension name are registered with the operating system. The Data Socket client uses the automation server when it detects the corresponding access method, which is not supported natively but is registered with the OS. Applications of Data Socket extensions include direct links to OPC servers, external instruments, Data Acquisition (DAQ) devices, databases, multimedia devices, and others.

Detailed Description Text (84):
Where the Data Socket client is a control, the user drops the Data Socket control, which is preferably a standard interface compliant control, onto a window of a container in step 302. An example of the window is a Visual Basic form. The

standard control interface is preferably the interface defined by the Microsoft ActiveX Control Specification. Controls which conform to the ActiveX OLE Control Specification are referred to as ActiveX controls. The control dropped in step 302 is preferably embodied within the National Instruments" "ComponentWorks" product, which comprises ActiveX controls. The ComponentWorks controls are described herein being used by the Microsoft Visual Basic development environment container. However, the ComponentWorks controls described may be employed in, but not limited to, the following list of containers: Microsoft Visual FoxPro, Microsoft Access, Borland Delphi, Microsoft Visual C++, Borland C++, Microsoft Internet Explorer, Netscape Navigator, Microsoft Internet Developer Studio, National Instruments HiQ, and any other container which supports the ActiveX control specification.

Detailed Description Text (108):
If the access method is "dstp" as determined in step 430, then in step 432 the Data Socket client attempts to make a connection to the Data Socket server identified by the URL using the host name or Internet address encoded in the URL according to standard URL syntax. As described above, the access mode "dstp" directs the Data Socket client to connect to the Data Socket server identified in the URL. If the connection is established in step 432, then in step 434 the Data Socket client sends a command indicating a request to subscribe to a specific tag, or to write the value of a specific tag maintained by the Data Socket server. The Data Socket client preferably sends this command over TCP/IP. If the specific tag does not exist on the server, then the server may create the tag and give it an initial value, or may report back an error indicating that that the requested tag does not exist. This is a configuration option on the Data Socket server. Reporting errors is preferably done by sending commands over the TCP/IP connection. Commands are preferably sequences of bytes sent over a TCP/IP connection.

Detailed Description Text (117):
FIG. 6 illustrates an example of reading a sound file or .wav file using the Data Socket tools of the present invention. In this example, a file called "chirp.wav" is stored as a standard Windows WAV file. In using the Data Socket, the user provides the application and/or the Data Socket client with a universal resource locator (URL) that identifies the file. The Data Socket tool reads the WAV file and converts the data into an array, which is displayed on the graph. More specifically, the Data Socket client uses the URL to find and open the .WAV file chirp.wav, which is a sound file. The Data Socket client opens the file, and the extension or plug-in converts the data directly into a one dimensional array of samples, which are the sample points of the array. This 1D array of samples is directly useable by the application. Without the Data Socket client, the user would be required to create code to open the file, read the header and determine how to parse the data, which would necessarily be specific to a sound file. With the Data Socket control the application is allowed to know nothing about file I/O and nothing about .wav files. The Data Socket client/tools can also be used to read multiple channels of data (stereo) stored in a file display, wherein each channel is, for example, separately plotted on the graph.

Detailed Description Text (122):
FIGS. 8 and 9 illustrate an example where the Data Socket client is configured as an ActiveX component and used in a Visual Basic application. FIG. 8 illustrates the Data Socket ActiveX control on a Visual Basic form, and FIG. 9 illustrates the corresponding Visual Basic code. In this example, the Data Socket client is used to reference the URL dstp://localhost/wave. As shown, in order to connect the Data Socket control to a data source or target, the user is only required to include code which calls one method specifying the data source using a URL and the connection type, either a read or write. The URL can specify a file, Web server, FTP server, or a Data Socket server connected to another DataSocket control in another application. The URL can specify other data sources, as desired.

Detailed Description Text (125):

In AutoUpdate mode, data is automatically exchanged between the Data Socket control and the data source or target whenever the data changes. In ReadAutoUpdate mode, the control fires an event when new data becomes available. In the event handler, the user can configure the application to process the new data as desired.

Detailed Description Text (148):
The present invention uses a protocol referred to as the Data Socket transfer protocol (DSTP) to communicate between the Data Socket tools and the Data Socket server. In the preferred embodiment, the DSTP is based on the industry-standard TCP/IP protocol using a registered user port number 3015. A brief description of the protocol used in the preferred embodiment follows:

Detailed Description Text (183):
Data Socket handles all tasks of converting data and data attributes from their native application format (strings, arrays, Booleans, etc.) into a TCP/IP suitable format, referred to as the Flex Data format, and converting back from the Flex Data format on the client end. Because the DSTP network communication only requires TCP/IP support, the Data Socket can be used to share information through many different types of networks, including the Internet. The Data Socket can be used to share information between machines located on opposite sides of the world using local Internet service providers. Of course, Data Socket and the Data Socket server can be used on a local Windows network or in a single stand-alone computer.

Previous Doc Next Doc Go to Doc#

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Generate Collection

L6: Entry 18 of 23 File: USPT Feb 24, 1998

DOCUMENT-IDENTIFIER: US 5721876 A
TITLE: Sockets application program mechanism for proprietary based application programs running in an emulation environment

Abstract Text (1):
A host data processing system operating under the control of a host operating system such as an enhanced version of the UNIX operating system on a RISC based hardware platform includes an emulator which runs as an application process for executing emulated system (ES) user application programs. The emulator includes a number of emulated system executive service components including a socket command handler unit and a socket library component operating in shared memory and an interpreter, an emulator monitor call unit (EMCU) and a number of server components operating in host memory. The host operating system further includes a host socket library interface layer (API) which operatively connects through a TCP/IP network protocol stack to the communications facilities of the hardware platform. The socket server components operatively connect ES TCP/IP application programs to the socket library interface layer of the host operating system when such application programs issue standard ES socket library calls. The socket command handler unit maps the ES socket library calls into appropriate input/output requests directed to the EMCU. The EMCU directs the requests to an appropriate socket server component which in turn issues the appropriate host socket library calls to the host socket interface layer thereby eliminating both the need to communicate through additional protocol stacks and to provide additional communication hardware facilities.

Application Filing Date (1):
19950330

Brief Summary Text (8):
With the advent of open system platforms which operate under the control of versions of the UNIX* operating system, it becomes more and more desirable to be able to efficiently run proprietary application programs on such open systems without having to rewrite or port such application programs. Also, when certain types of proprietary application programs are written to utilize standard communication network protocols, such as TCP/IP, implemented as part of the proprietary operating system, this may complicate the process of running these programs in an open system environment.

Brief Summary Text (10):
One approach which has been considered is to provide a separate TCP/IP protocol stack and separate hardware facilities for servicing the network demands of such proprietary application programs. While this approach appears satisfactory, it creates considerable processing delays causing such proprietary application programs to run too slow thereby reducing overall system performance. This can be a substantial disadvantage particularly when such programs are to be executed in an emulator environment. Also, this approach is too costly in terms of memory resources and is unable to take direct advantage of the facilities of the open system environment.

Brief Summary Text (15):

The host operating system further includes a socket interface layer which operatively connects through a TCP/IP network protocol stack to the communications facilities of the host hardware platform. The hardware platform operatively couples to conventional network facilities. Socket server components operatively connect ES TCP/IP application programs to the socket library interface layer of the host operating system in response to standard ES socket library calls issued by such programs. The socket command handler unit contains specific routines which map the ES socket library calls into appropriate input/output requests directed to the EMCU which in turn directs the requests to a main socket server component. The socket server component in turn issues the appropriate host socket library calls to the host socket library interface layer thereby eliminating both the need to communicate through additional protocol stacks and the need to provide additional communication hardware facilities.

Brief Summary Text (17):
In the preferred embodiment of the present invention, the management of the different socket operations being executed is carried out using a socket control table. According to the present invention, the socket control table contains a number of addressable slot locations which are allocated to user applications on a first come first serve basis. The address of each assigned slot location is used as an index to the control table for obtaining information pertaining to the actual socket (number) assigned by the network facilities. The socket control table address is returned to the application which issued the library call and is used by the application as the assigned socket number.

Brief Summary Text (18):
Each socket control table slot includes a number of fields which when the slot is assigned, store information pertinent to the assigned socket. For example, the fields include a first field for storing the actual assigned socket number, a second field for indicating the owner/creator of the socket entry (i.e. main or child socket server process) and a number of fields for storing pipe descriptor information to establish and maintain interprocess communications between parent and child socket server processes. The control table arrangement enables the efficient multiplexing of socket library requests by server components thereby increasing overall system performance. The size of the socket control table is selected to be large enough to accommodate a substantial number of concurrent socket operations.

Brief Summary Text (19):
Overall, the socket mechanism of the present invention allows proprietary application programs running in the emulation environment access to host TCP/IP protocol stack communication facilities of the host enhanced UNIX operating system thereby eliminating the need to communicate through additional protocol stacks or to provide additional communication hardware facilities. This in turn enhances overall system performance as well as eliminating the need for having to allocate additional system resources (e.g. memory).

Drawing Description Text (5):
FIGS. 4a and 4b are diagrams illustrating, in greater detail, the socket control table structure of FIG. 1.

Detailed Description Text (15):
The present invention includes an ES socket command handler unit 284 and ES socket library 286. The ES socket library 286 is constructed to provide the same socket application program interface (API) as provided in the emulated system. This interface is described in detail in the manual entitled "GCOS 6 HVS TCP/IP SOCKET API FOR C USERS" published by Bull HN Information Systems, Inc., copyright 1993, order no. RD89-00.

Detailed Description Text (18):

As indicated in FIGS. 1a and 1b, the next layer within the user level is the emulator executive level 68. This level includes certain components present in the emulated system which have been transformed into new mechanisms which appear to the remaining unchanged components to operate as the original unchanged components of the emulated system. At the same time, these new mechanisms appear to the components of the kernel level 64 as native components with which the host system is accustomed to operate. As shown, the components include the interpreter 72, an emulator monitor call unit (EMCU) 73, dynamic server handler (DSH), main socket server component 98, a number of child socket processes 96 and a socket control table 94 operatively coupled together as shown.

Detailed Description Text (23):
The dynamic server handler (DSH) 92 is created by EMCU 73 during initialization. The server 92 communicates with emulated system processes through MQI 84 as indicated. The lower level main socket server 98 and socket control table 94 are dynamically created by higher level server 92 for carrying socket operations according to the present invention. The main socket server 98 creates child socket processes as a function of the type of socket operation to be performed and manages such child processes through socket control table 94. All of the servers operate as root and therefore have super user privileges with access to any file within the host system 54.

Detailed Description Text (27):
In the preferred embodiment, the kernel/operating system level 64 further includes as an interprocess communications facility, an implementation of "sockets" which includes a host sockets library 97 for storing a plurality of socket subroutines and network library subroutines and a TCP/IP network protocol stack facility 99 arranged as shown. The stack facility 99 connects to an Ethernet driver included within kernel level 64 (not shown) which communicates with the Ethernet LAN 58c.

Detailed Description Text (28):
As indicated in the system of FIGS. 1a and 1b, as in the case of the AIX operating system, the socket subroutines contained in host sockets library 97 serve as the application program interface (API) for TCP/IP. This API provides three types of communications services which use different components of TCP/IP. These are reliable stream delivery, connectionless datagram delivery and raw socket delivery. For further information regarding sockets, reference may be made to various well known publications and texts such as publications of IBM Corporation describing the AIX Version 3.2 for RISC System/6000 and the text entitled "UNIX System V Release 4: An Introduction for New and Experienced Users" published by Osborn McGraw-Hill, Copyright 1990 by American Telephone and Telegraph Company.

Detailed Description Text (30):
In the emulated system, several different types of applications included within block 22 issue ES socket library calls to carry out read and write operations between the host and remote computer systems using TCP/IP. In the preferred embodiment, these applications include an FTP interactive program which allows a user to transfer files between the system of FIGS. 1a and 1b and a remote system; a Telnet interactive program which implements the UNIX remote terminal logon and a remote file access (RFA) which gives a user access to the facilities of a remote system. For specific information about these types of applications such as FTP, reference may be made to the publication entitled "GCOS 6 HVS TCP/IP Reference Manual" published by Bull HN Information Systems Inc., copyright 1993, Order Number RE86-01.

Detailed Description Text (35):
As indicated in FIG. 2, the RQIO request is forwarded to the socket server 98 for processing. The socket server 98 obtains the request by reading an assigned MQI socket queue (block 208) as described herein. It examines the IORB and determines from the device specific word (dvs) containing the actual socket library call if

the function/operation specified in the call is a blocking function. That is, it determines if the socket operation will require substantial time to execute so as to prevent socket server 98 from responding to socket function requests issued by other application programs 22 thereby impairing system performance. If it is not a blocking function (block 210) (i.e., will not incur substantial delay), then socket server 98 performs the designated socket operation (block 212) using the host TCP/IP facilities 99, posts the IORB and returns control back to the handler/user application program (block 214).

Detailed Description Text (40):
The socket function is applied as an input to the ES socket library 286 and results in the generation of the MCL 3800 monitor call as in the emulated system. This ensures that application program 22 sees the same interface in FIG. 1 as in the emulated system. The MCL 3800 monitor call is applied to the socket monitor call handler unit 284 which locates the corresponding function as indicated in block 202 of FIG. 2. As in the emulated system, the major function code high order byte value "38" through a first level table branching operation causes control to be passed from the executive MCL handler to the TCP/IP MCL handler of block 284. Using the minor function code low order byte value "00" contained in the MCL 3800 monitor call, the TCP MCL handler via a second level table branching operation locates the appropriate socket handler routine which in the instant case is "socket".

Detailed Description Text (47):
Creation of Socket Control Table Structure

Detailed Description Text (48):
The main socket server 98 creates an addressable socket control table structure 94 in host system space as part of the initialization function. The socket control table structure 94 is shown in detail in FIGS. 4a and 4b. The table 94 is set up to have a size of addressable 1024 slots or locations for storing information pertaining to a number of sockets. The first three address slots of FIGS. 4a and 4b are reserved for convenient storage. Thus, the socket address values start with an index value of "3".

Detailed Description Text (49):
As indicated in FIGS. 4a and 4b, each of the socket locations contains the following fields: sockno, sock.sub.-- pid, sock.sub.-- pid.sub.-- xmit, sock.sub.-- flags, sock.sub.-- wrpipefd, sock.sub.-- xwrpipefd, sock.sub.-- clspipefd0 and sock.sub.-- clspipefd1. The sockno field is used to store the actual or real socket number returned by the host tcp/ip facility 99. This is done to prevent issuing duplicate sockets/processes for handling application program requests.

Detailed Description Text (53):
The main socket server 98 initializes the required ES system data structures (e.g. SCB, NCB) and the socket control table pointers (i.e. initial, current and sockmax) used to access socket control table 94 to the appropriate initial values. The server 98 also issues a host kernel call to obtain its process ID.

Detailed Description Text (55):
The host library component 97 invokes the host TCP/IP network protocol stack facility 99 which initiates in a conventional manner, the appropriate series of operations for obtaining the actual socket number from the TCP/IP network facility. As indicated by block 312, the server 98 determines if it received a socket number. When it has received a socket number, it then finds the first unused slot in the socket control table 94 by accessing the table with the socket current table pointer. It then stores in the sockno field of the slot, the descriptor value returned by the TCP/IP stack facility 99 which corresponds to the actual socket number (block 316 of FIG. 3a). It also sets the main.sub.-- sock indicator flag of the slot for indicating that the socket is owned by the socket server 98.

Detailed Description Text (56):
Since this is the first socket operation, the slot location identified by an address (index) of 3 is the first available slot in socket control table 94. It is this value that is returned to the user application program as the socket while the actual socket descriptor (actual socket number) remains stored in the sockno field of slot 3. As indicated in block 318 of FIG. 3a, server 98 stores the address index value 3 in the dv2 field of the IORB structure and terminates the RQIO by posting the IORB with good status (block 318). If no actual socket was obtained, then socket server 98 terminates the RQIO and returns error status (block 314).

Detailed Description Text (68):
As indicated by block 342 of FIG. 3b1, the parent socket server process 98 returns to process the next incoming socket IORB. The child process 96 begins initial processing by executing the operations of block 344. As indicated, the child process 96 first locates the slot entry (i.e. index of 3) in the socket control table 94 for the file descriptor provided as an argument by the IORB and stores the child process in the sock.sub.-- pid field of the entry for subsequent proper closing of the socket. Also, the child process 96 marks the socket as being in the accept blocked state by setting the in.sub.-- accept indicator flag to the appropriate state. This enables the server to break-in and stop the operation.

Detailed Description Text (70):
The child process 96 loops waiting for the return of the actual socket number from the TCP/IP stack facility 99. If there is an error, the result is that an error is entered into the device status word of the IORB data structure and the RQIO operation is terminated (block 350). Assuming that there was no error, the child process 96 performs the series of operations of block 352. As indicated, the child process 96 resets the state of the in.sub.-- accept indicator flag in the third slot of the socket control table 94 corresponding to index value of 3 in addition to saving the child process ID (pid) in the sock.sub.-- pid field of that slot.

Detailed Description Text (71):
Next, the child process 96 finds the next unused slot in the socket control table 94 which corresponds to the slot location having an index of 4 by accessing the table with the socket current pointer value after being incremented by one. It then stores in the sockno field of the slot, the descriptor value returned by the TCP/IP stack facility 99 which corresponds to the actual socket number (e.g. 100). It also sets/verifies that the state of the main.sub.-- sock indicator (i.e. "0") designates that the socket is owned by the child process. Also, it saves the child process ID in the sock.sub.-- pid field of the slot having the index of 4. At this point, there are two slots which contain the pid of the child process 96. It will be noted that while the child process started with the slot having index of 3 in socket control table 94, it now is operating with the slot having an index of 4. While this example uses sequentially numbered slots, it will be appreciated that the slot to which the child process 96 sequences may be any slot within the socket control table 94 which corresponds to the next available slot location.

Detailed Description Text (82):
The socket server process 98 upon receipt of the request I/O performs the operations of block 384. That is, it translates the HVS address arguments from the ES system space into host space as required. Using the IORB socket descriptor index value, the server process 98 locates the socket control table slot specified by that socket descriptor. The socket server process 98 then determines if a child process exists for the socket (block 386) by examining the indicator flag main.sub.-- sock of the slot. If the sock.sub.-- pid and main.sub.-- sock indicator flag are set to values indicating that a child process already exists, then the server process 98 performs the operations of block 388. This involves obtaining the IPC pipe descriptor stored in the sock.sub.-- wrpipefd field of the slot identified by the index value, setting up a message buffer for the operation and sending a message to the child process via the IPC pipe. Also, the server process 98 returns

to process the next available IORB request.

Detailed Description Text (85):
Next, the spawned child process enters a socket receive routine (sorecv) which reads the newly requested socket operation (message) contained in the IPC pipe (block 394) using read file descriptor pipefd[0]. It checks for the presence of an ioctl command specifying a break-in. Assuming that the IPC pipe contains no ioctl command, the child process issues a socket receive library call to the host system. That is, the child process issues the socket receive (recv0) call to the host socket library (block 402) which contains the actual socket number obtained from socket control table 94.

Detailed Description Text (86):
Next, the child process enters a receive loop wherein it determines if the receive is completed at which time it sends back the data to the ES application and if the data was correctly received (block 404). If it was, the child process terminates the receive RQIO and posts the IORB with good status (block 406). As indicated in FIG. 3c2, the child process following the completion of the first receive again performs another read PIPE operation and processes the next receive request. Thus, the child process continues to handle receive requests on that socket until the socket is closed. Accordingly, there may be a series of receive socket requests issued before the child process receives a ioctl command specifying a break-in. If the child process receives such a command, it performs the operations of blocks 398 and 400. It terminates the receive socket operation taking place at that time according to the state of the in.sub.-- rcbi indicator in the flags field of the socket control table slot associated with the socket. It then terminates the RQIO request and posts good status in the IORB (block 398).

Detailed Description Text (87):
From the above, it is seen that the socket mechanism of the present invention performs an ES socket library receive function by spawning a child process which prevents blocking socket server process 98. Through the use of socket control table 94, the socket server process 98 is able to communicate and efficiently manage the operations of the child process which in turn communicates with the host library and TCP/IP network protocol stack facilities.

Detailed Description Text (90):
For further more specific details regarding the implementation of the socket mechanism of the present invention, reference may be made to the source listings contained in the attached appendix which include the following items: Main Socket Server Component; Socket Handler Component; Socket Control Table Structure; Socket Call and IORB Related Data Functions; and Socket Control and Related Data Structures.

CLAIMS:

1. A host system having a memory organized into shared and host memory areas and a hardware platform operatively coupled to a communications network for communicating with other units using a communications network protocol, said host system emulating the execution of emulated system instructions by an emulator running as an application process on said host system, said emulator including a number of emulated system executive service components operating in said shared memory area comprising a command handler unit and an interpreter, an emulator monitor call unit (EMCU) and server facilities operating in said host memory area, said host system further including operating system facilities for providing a number of services for host programs, said operating system facilities being coupled to said communications network and to said EMCU, said host system further including a socket mechanism for processing socket library calls generated by emulated system application programs running under control of said emulator, said socket mechanism comprising:

socket command handler means included within said command handler unit, said socket command handler means in response to each socket library call specifying a socket function received from an emulated application program generating an I/O request containing an input/output (IORB) data structure coded to contain a socket command identifying a socket operation mapped from said each socket library call and forwarding said request to said EMCU for issuance to said server facilities;

an addressable socket control table located in said host memory area, said socket control table having a number of locations for storing predetermined formatted control words generated in response to socket commands generated by said socket command handler means specifying socket operations designated by said socket commands; and,

said server facilities including main socket server processing means, said main socket server processing means in response to each socket command selectively generating a child process as a function of the length of time required to execute that type of socket operation on said communications network through the operating system facilities and for storing and updating said control words stored in said socket control table for executing socket operations initiated by socket library calls generated by any one of said emulated system application programs.

5. The host system of claim 4 wherein said main socket server processing means generates said socket control table to contain a predetermined number of locations for processing a maximum number of socket operations for said number of emulated system application programs.

7. The host system of claim 6 wherein said protocol stack implements said predetermined protocol which corresponds to a TCP/IP protocol.

8. The host system of claim 1 wherein said control words of said socket control table are cleared from said socket control table upon the completion the use of assigned socket network resources by corresponding ones of said emulated system application programs.

11. The host system of claim 1 wherein said socket control table includes socket pointer address means for accessing said locations for storing and updating said control words as required for executing operations specified by said socket commands, said main socket processing means in response to a first socket command and upon receipt of a socket number value from said communication network storing said socket number value corresponding to an actual socket number into a first available location in said socket control table designated by said socket pointer address means and returning to one of said emulated system application programs which issued said socket command, an index value obtained from said socket pointer address means identifying said first available location in said socket control table for use in subsequent socket operation requests.

13. The host system of claim 11 wherein said one of said emulated system application programs issues a subsequent socket command which includes said index value specifying a type of socket operation requiring more than a predetermined time period, said main server processing means being operative to access said control word stored in said location specified by said index value for obtaining the actual socket number to be used in issuing socket library calls to said communications network and to advance said socket pointer address means to specify a next available location in said socket control table.

20. The host system of claim 14 wherein each control word stored in said socket control table remains in said table for a period of time that the actual socket is being used by one of said emulated system application programs.

21. The host system of claim 14 wherein said subsequent socket command specifies an accept type operation, said child process accesses a designated control word location of said socket control table specified by said index value, sets said child process ID in said second number of fields and sets an indicator flag of said third number of fields to indicate that said socket is in an accept blocked state and generates a new socket call to said host requesting a new socket.

22. The host system of claim 21 wherein said child process upon receipt of an actual socket number from said communications network in response to said new socket call, first accesses said socket control table for resetting said indicator flag removing said blocked state and then accesses a next available location of said socket control table for storing in said first field of said control word, the actual socket number of said new socket returned by said host communications network and write pipe descriptor value in a first one of said second number of fields for enabling said child process to communicate with said main server processing means for processing subsequently issued socket commands.

23. A method of organizing an emulator for executing socket library calls received from a plurality of emulated system application programs running under control of a host operating system on a host system connected to host communications network facilities which maximizes the use of said communications network facilities, said operating system including a host socket library, a host protocol stack operatively coupled to said library and to said host communications network facilities, said host system having a memory organized into shared and host memory areas and a hardware platform operatively coupled to said host communications network facilities for communicating with other units using a communications network protocol, said emulator including a number of emulated system executive service components operating in said shared memory area comprising a command handler unit operatively coupled to an emulated system socket library and an interpreter, an emulator monitor call unit (EMCU) coupled to said operating system and server facilities operating in said host memory area, said host system further including a socket mechanism for processing socket library calls generated by emulated system application programs running under control of said emulator, said method comprising the steps of:

including socket command handler means included within said command handler unit;

generating an I/O request containing an input/output (IORB) data structure by said socket command handler means in response to each socket library call specifying a socket function received from an emulated application program;

mapping said each socket library call into a socket command coded to identify a socket operation for inclusion in said I/O request;

forwarding said FO request to said EMCU for issuance to said server facilities;

storing an addressable socket control table in said host memory area, said socket control table having a number of locations for storing predetermined formatted control words generated in response to socket commands generated by said socket command handler specifying socket operations designated by said socket commands;

including a main socket server processing means in said server facilities;

selectively generating a child process as a function of the length of time required to execute that type of socket operation on said communications network through the operating system facilities by said main socket server processing means in response to said each socket command; and,

storing and updating said control words stored in said socket control table by said child process and main socket server processing means for executing socket

operations initiated by socket library calls generated by different ones of said emulated system application programs.

27. The method of claim 26 wherein said protocol stack implements said network protocol which corresponds to a TCP/IP protocol.

28. The method of claim 23 wherein said method further includes the step of clearing control words from said socket control upon the completion the use of assigned socket network resources by corresponding ones of said emulated system application programs.

31. The method of claim 23 wherein said socket control table includes socket pointer address means for accessing said locations for storing and updating said control words as required for executing operations specified by said socket commands, said method further including the steps of:

storing said socket number value corresponding to an actual socket number into a first available location in said socket control table designated by said socket pointer address means main in response to a first socket command and upon receipt of a socket number value from said communication network; and,

returning to one of said emulated system application programs which issued said socket command, an index value obtained from said socket pointer address means identifying said first available location in said socket control table for use in subsequent socket operation requests.

33. The method of claim 31 wherein said one of said emulated system application programs issues a subsequent socket command which includes said index value specifying a type of socket operation requiring more than a predetermined time period, said method further including the steps of:

accessing said control word stored in said location specified by said index value for obtaining the actual socket number to be used in issuing socket library calls to said communications network; and,

advancing said socket pointer address means to specify a next available location in said socket control table.

40. The method of claim 34 wherein each control word stored in said socket control table remains in said table for a period of time that the actual socket is being used by one or said emulated system application programs.

41. The method of claim 34 wherein said subsequent socket command specifies an accept type operation, said method further includes the steps of:

accessing a designated control word location of said socket control table specified by said index value by said child process;

setting said child process ID in said second number of fields and an indicator flag of said third number of fields to indicate that said socket is in an accept blocked state; and,

generating a new socket call to said host socket library requesting a new socket.

42. The host system of claim 41 wherein said method further includes the steps of:

first accessing said socket control table by child process upon receipt of an actual socket number from said communications network in response to said new socket call for resetting said indicator flag removing said blocked state; and,

accessing a next available location of said socket control table for storing in said first field of said control word, the actual socket number of said new socket returned by said host communications network and write pipe descriptor value in a first one of said second number of fields for enabling said child process to communicate with said main server processing means for processing subsequently issued socket commands.

Previous Doc Next Doc Go to Doc#

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Generate Collection

L6: Entry 21 of 23 File: USPT Jul 16, 1996

DOCUMENT-IDENTIFIER: US 5537417 A
TITLE: Kernel socket structure for concurrent multiple protocol access

Application Filing Date (1):
19930129

Brief Summary Text (4):
However, there are many vendors who have developed their own hardware and software solutions for integrating multiple computer systems. In particular, they have developed different ideas of the format and protocols that should be followed in transporting data through the networks. Some protocols support expedited data transmission by bypassing the standard data flow controls; others require all data to go through the controls. Specialized protocols are used for transport tasks such as establishing and terminating connections between computer systems. Examples of well known communication protocols include System Network Architecture (SNA), Digital Network Architecture (DECnet), Transmission Control Propotocol/Internet Protocol (TCP/IP), Network Basic Input/Output System (NetBIOS), Open Systems Interconnection (OSI) and AppleTalk. Other protocols are known and widely used.

Drawing Description Text (5):
FIG. 3 is a diagram of a conventional socket control block.

Drawing Description Text (6):
FIG. 4 is a diagram of the socket control block of the present invention.

Detailed Description Text (4):
FIG. 1 shows three single protocol networks 10, 12, 14 are interconnected through a gateway 59 built according to the principles of the present invention. With the growth of network distributed environments, it is not uncommon to see networks using four or five different protocols, such as TCP/IP, NetBIOS, SNA or AppleTALK. Since applications which run on one network will not often run with applications on the other, transport of data throughout the network is hindered. As discussed above, the MultiProtocol Transport Network (MPTN) 16 as taught in the above referenced patent addresses these problems by defining an interface and a compensation mechanism to a set of transport services that provide connections across a plurality of transport protocols. By providing a transport boundary between the networks and the applications resident on systems, MPTN provides a common transport interface so which messages from the applications can be transported over any protocol in the network.

Detailed Description Text (5):
As shown in FIG. 1, hosts 20 and 22 are coupled to the multiprotocol transport network 16. While the MPTN 16 appears as though it is a single logical network having a single protocol, host 20 is coupled to a first network 10 which communicates via protocol X, e.g., TCP/IP, and host 22 is coupled to a second network 12 which communicates via protocol Y, e.g., NetBIOS.

Detailed Description Text (7):
FIG. 2 depicts the code modules in memory at a computer system which is coupled to

the multiprotocol transport network in greater detail. The socket programming interface 60 and common transport semantics 62 correspond to the socket layer and separate the applications 64, 66, 68 from the service drivers 70, 72, 74. Three types of applications 64, 66, 68 are shown in the application layer. To utilize the socket structure of the present invention, NetBIOS applications would be rewritten to the socket API to become the NetBIOS socket application 66. The standard local IPC socket application 64 and TCP/IP applications 68 are already written to a standardized socket programming interface and so would require an minimum of revisions. The common transport semantics 62, include the socket control blocks, which will be described in greater detail below. An interface between the socket layer and the transport layer is comprised by the local IPC service driver 70, the NetBIOS service driver 72 and the INet service driver 74 which correspond to the applications in the application layer. The service drivers are used to emulate the common transport semantics for the transport protocol drivers in the transport layer. In the present invention, they may also contain the compensating means described in the above referenced application which converts a message intended for a first protocol by the application to a second protocol supported by the network. The transport layer includes the NetBIOS 76 and TCP/IP 78 protocol drivers which cause the application message to confirm to the protocol format. There is no corresponding local IPC module as it describes a local protocol whose messages are not placed on the network. The network and network interface layers are not pictured, the latter would include device drivers for the I/O hardware which couples the computer system to the network, e.g., a token ring or ethernet driver; the former might include drivers written to the well known NDIS interface.

Detailed Description Text (10):
In the present invention, the socket interface is used to connect between replicas of a distributed application or the client and server portions of a client/server applications using a variety of transport protocols. The application can select the transport protocol or request that the socket layer determine the protocol. With the non-native networking feature of the present invention, applications written to communicate with one another using one protocol can chose to communicate on another transport protocol which might be optimized for the network environment. For example, an application written for TCP/IP could communicate the NetBIOS protocol over the network, thus, give the distributed application a significant performance gain.

Detailed Description Text (11):
A conventional socket control block is depicted in FIG. 3, a socket control block 100 contains information about the socket, including send and receive data queues, their type, the supporting protocol, their state and socket identifier. The socket control block 100 includes a protocol switch table pointer 104 and a protocol control block pointer 106. The pointers are used to locate the protocol switch table (not pictured) and the protocol control block 102 respectively. The protocol switch table contains protocol related information, such as entry points to protocol, characteristics of the protocol, certain aspects of its operation which are pertinent to the operation of the socket and so forth. The socket layer interacts with a protocol in the transport layer through the protocol switch table. The user request routine PR.sub.-- USRREQ is the interface between a protocol and the socket. The protocol control block contains protocol specific information which is used by the protocol and is dependent upon the protocol.

Detailed Description Text (12):
A socket control block according to the present invention is shown in FIG. 4. Here, the socket control block 110 is shown broken into two sections, the main socket control block which is largely identical to the conventional socket control block above and the MPTN extension 112 which contains many of the features necessary for the present invention. Both are linked together by a multiprotocol transport network extension 113. Each of the protocols which could be potentially used to send or receive data is sent a request to create a protocol control block 120, 122

at the time the new socket is created. After a selection procedure, new socket then contains information regarding each selected protocol the protocol switch table pointer 114, 118, the protocol control block pointer 116, 119, and a pointer to a interface address if applicable for the particular protocol (not pictured). As above, the protocol switch table pointer refers to a respective protocol switch table which defines various entry points for that protocol. Also, the protocol control blocks 120, 122 contain protocol specific information.

Detailed Description Text (15):
A sample socket control block structure is given in the code below:

Detailed Description Text (16):
Conventional socket creation starts with a call to the socket API. A domain table is searched for the address family, the type and protocol which is desired by the application. If a match is found tile protocol switch table entry is set in the socket control block. Next, the user requests entry and the selected protocol is called to create the protocol control block. If a match is not found in the domain table for the address family, type and protocol desired by the application, an error is returned to the application.

Detailed Description Paragraph Table (1):

```
_____ __ /*
socketva.h..... * Kernel structure per socket. * Contains send and receive buffer
queues, * handle on protocol and pointer to protocol * private data, error
information and MPTN extensions. * The first part of this structure ( upto the
so.sub.-- mptn field ) is identical * to the BSD4.3 socket control block. */ struct
socket { short so.sub.-- type; /* generic type, see socket.h */ short so.sub.--
options; /* from socket call, see socket.h */ short so.sub.-- linger; /* time to
linger while closing */ short so.sub.-- state; /* internal state flags SS.sub.-- *,
below */ caddr.sub.-- t so.sub.-- pcb; /* protocol control block */ struct protosw
far *so.sub.-- proto; /* protocol handle */ /* * Variables for connection queueing.
* Socket where accepts occur is so.sub.-- head in all subsidiary sockets. * If
so.sub.-- head is 0, socket is not related to an accept. * For head socket so.sub.-
- q0 queues partially completed connections. * while so.sub.-- q is a queue of
connections ready to be accepted. * If a connection is aborted and it has so.sub.--
head set, then * it has to be pulled out of either so.sub.-- q0 or so.sub.-- q. *
We allow connections to queue up based on current queue lengths * and limit on
number of queued connections for this socket. */ struct socket far *so.sub.--
head; /* back pointer to accept socket */ struct socket far *so.sub.-- q0; /* queue
of partial connections */ struct socket for *so.sub.-- q; /* queue of incoming
connections */ short so.sub.-- q0len /* partials on so.sub.-- q0 */ short so.sub.--
qlen; /* number of connections on so.sub.-- q */ short so.sub.-- qlimit; /* max
number queued connections */ short so.sub.-- timeo; /* connection timeout */
u.sub.-- short so.sub.-- error; /* error affecting connection */ short so.sub.--
pgrp; /* pgrp for signals */ u.sub.-- long so.sub.-- oobmark; /* chars to oob mark
*/ / * Variables for socket buffering. */ struct sockbuf { u.sub.-- long sb.sub.--
cc; /* actual chars in buffer */ u.sub.-- long sb.sub.-- hiwat; /* max actual char
count */ u.sub.-- long sb.sub.-- mbcnt; /* chars of mbufs used */ u.sub.-- long
sb.sub.-- mbmax; /* max chars of mbufs to use */ u.sub.-- long sb.sub.-- lowat; /*
low water mark (not used yet) */ struct mbuf far *sb.sub.-- mb; /* the mbuf chain
*/ struct proc far *sb.sub.-- sel; /* process selecting read/write */ short
sb.sub.-- timeo; /* timeout (not used yet) */ short sb.sub.-- flags; /* flags, see
below */ } so.sub.-- rcv, so.sub.-- snd; /* MPTN SOCKET EXTENSION */ struct m.sub.-
- esock far * so.sub.-- mptn; /* socket MPTN extensions */ #define SB.sub.-- MAX
((u.sub.-- long)(64*1024L)) /* max chars in sockbuf */ #define SB.sub.-- LOCK
0x01 /* lock on data queue (so.sub.-- rcv only ) */ #define SB.sub.-- WANT 0x02 /*
someone is waiting to lock */ #define SB.sub.-- WAIT 0x04 /* someone is waiting for
data/space */ #define SB.sub.-- SEL 0x08 /* buffer is selected */ #define SB.sub.--
COLL 0x10 /* collision selecting */ }; /* Socket extension for MPTN * The socket
control block points to this structure which contains * all the MPTN related socket
```

```
extensions. * Since m.sub.-- esock, the MPTN extensions to sockets, is contained in
one * mbuf, we could use the rest of mbuf space to accommodate the pcb * pointers.
*/ /* defines the structure for storing additional pointers. * used in m.sub.--
esock. * The structure m.sub.-- addr defines the address format.It is defined in
mptndef.h. * The structure m.sub.-- info defines the user specified connection
characteristics * and is defined in mptndef.h. * The structure bnlst defines the
user configured protocol preference list * and is defined in mptndef.h. * */ struct
m.sub.-- sopcb { struct protosw far * so.sub.-- proto; /* protocol switch ptr */
caddr.sub.-- t so.sub.-- pcb; /* pointer to the PCB */ struct m.sub.-- addr far *
so.sub.-- bnsap; /* the network interface that * the PCB refers to */ }; struct
m.sub.-- conn.sub.-- stat { char type; /* DST.sub.-- BNSAP.sub.-- FOUND,USE.sub.--
PREF.sub.-- LIST,CACHE. * GW.sub.-- NEXT.sub.-- HOP, GW.sub.-- BNSAP */ char
index; /* current network...as an index in to the * pref. list as the case is. */
caddr.sub.-- t ftnam; /* destination name..for ABM case, Gateway * cases, it is
dst.sub.-- b.sub.-- nam. * In the native case or cache case,contains * the dest A-
addr */ struct bnlst far *prflst; /* pointer to the pref list */ }; struct m.sub.--
esock { int (* so.sub.-- upcall ) ( ); /* user upcall address */ struct socket far
* so.sub.-- relay; /* relay socket pointer:for gateway only */ struct m.sub.-- info
far * so.sub.-- info; /* connection chars given at m.sub.-- create( )*/ struct
m.sub.-- info far * so.sub.-- cinfo; /* conn.char of a connection */ short so.sub.-
- domain; /* user specified domain */ struct m.sub.-- addr far * so.sub.--
lanam; /* local user name */ struct m.sub.-- addr far * so.sub.-- panam; /* peer
user name */ struct mbuf far * so.sub.-- ctdata; /* connection/termination data */
struct mbuf far * so.sub.-- expdat; /* expedited data */ struct socket far *
so.sub.-- next; /* linked list of nonnative scbs.to search * for user-names*/
unsigned so.sub.-- mptn.sub.-- flag; /* mptn flags having the following defs*/
struct m.sub.-- conn.sub.-- stat conn.sub.-- stat; /* to maintain mptn.sub.--
connect status */ short so.sub.-- pcbnum ; /* number of pcbs currently in sopcb[ ]
*/ long so.sub.-- pcbstat; /* pcb state:bit position corresponds to * pcb array;
0=>in-use; else not-in-use*/ struct m.sub.-- sopcb sopcb [MPTN.sub.-- MAXPCB];/*
array of m.sub.-- sopcb structures */ #define MPTN.sub.-- NONNATIVE 0x01 /*
indicates nonnative connection */ #define MPTN.sub.-- NATIVE 0x02 /* indicates
native connection */ #define MPTN.sub.-- SO.sub.-- BIND 0x04 /* sock mptn
state=>addr bound */ #define MPTN.sub.-- SO.sub.-- UNBIND 0x08 /* sock mptn
state=>addr not bound*/ #define MPTN.sub.-- SO.sub.-- CONN 0x10 /* sock mptn
state=>connect req is made*/ #define MPTN.sub.-- SO.sub.-- NOMORE.sub.-- CONN
0x20 /* =>there will be no more conn. re-try * over other networks/protocols*/
#define MPTN.sub.-- SO.sub.-- LISTEN 0x40 /* init state of a passive sock */
#define MPTN.sub.-- SO.sub.-- CON.sub.-- HEAD.sub.-- WAIT 0x80 /* native con set
up..waiting for MPTN * connect header */ #define MPTN.sub.-- SO.sub.-- CON.sub.--
HEAD.sub.-- RCVD 0x0100 /* passive node...mptn con rcvd..*/ #define MPTN.sub.--
SO.sub.-- CON.sub.-- HEAD.sub.-- SENT 0x0200 /* Active node...mptn con sent ..*/
#define MPTN.sub.-- SO.sub.-- CON.sub.-- ESTABLISHED 0x0400 /* Connection
established ...*/ #define MPTN.sub.-- SO.sub.-- CLOSED 0x0800 /* local socket close
issued */ #define MPTN.sub.-- SO.sub.-- BCAST.sub.-- RCV 0x1000 /* the socket is
enabled to rcv * broadcast dgms.*/ #define MPTN.sub.-- SO.sub.-- IN.sub.--
ADDR.sub.-- ANY 0x2000 /* the socket is bound to in.sub.-- addr.sub.-- any */ }; /*
* Socket state bits. */ #define SS.sub.-- NORDREF 0x001 /* no file table ref any
more */ #define SS.sub.-- ISCONNECTED 0x002 /* socket connected to a peer */
#define SS.sub.-- ISCONNECTING 0x004 /* in process of connecting to peer */ #define
SS.sub.-- ISDISCONNECTING 0x008 /* in process of disconnecting */ #define SS.sub.--
CANTSENDMORE 0x010
```

Previous Doc Next Doc Go to Doc#